№ 1

Вот как можно удалить определенное число из массива в Swift:

swift

var arr = [1, 2, 3, 4, 5]

let numToRemove = 3

if let index = arr.firstIndex(of: numToRemove) {

arr.remove(at: index)

}

print(arr) // Output: [1, 2, 4, 5]

В этом примере мы используем метод firstIndex(of:), чтобы найти индекс первого вхождения числа, которое мы хотим удалить. Затем мы используем метод remove(at:), чтобы удалить элемент по найденному индексу.

№ 2

Если число повторяется несколько раз в массиве и вы хотите удалить все его вхождения, вы можете использовать метод removeAll(where:). Вот как это сделать в Swift:

swift

var arr = [1, 2, 3, 4, 3, 5, 3]

let numToRemove = 3

arr.removeAll { $0 == numToRemove }

print(arr) // Output: [1, 2, 4, 5]

Этот код удалит все вхождения числа 3 из массива arr. Мы передаем замыкание в метод removeAll, которое возвращает true для всех элементов, которые нужно удалить.

**Задача с перемещением 0 в конец массива**

class Solution {

func moveZeroes(\_ nums: inout [Int]) {

var nonZeroIndex = 0

// Переместим все ненулевые элементы в начало массива

for num in nums {

if num != 0 {

nums[nonZeroIndex] = num

nonZeroIndex += 1

}

}

// Заполним оставшуюся часть массива нулями

while nonZeroIndex < nums.count {

nums[nonZeroIndex] = 0

nonZeroIndex += 1

}

}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задача с выведением числа, с самым частым повторением**:

Для достижения временной сложности O(1), мы можем воспользоваться алгоритмом Мура для поиска большинства элемента в массиве. Этот алгоритм позволяет найти элемент, который встречается более чем n/2 раз, где n - это размер массива.

Алгоритм Мура работает следующим образом:

Инициализируем переменные candidate и count. На каждой итерации массива:

Если count равно 0, мы делаем текущий элемент кандидатом.

Если текущий элемент равен кандидату, мы увеличиваем count, в противном случае мы уменьшаем count.

В конце процесса candidate будет элементом, который встречается больше, чем n/2 раз, если такой элемент существует.

Вот реализация на Swift:

class Solution {

func majorityElement(\_ nums: [Int]) -> Int {

var candidate: Int?

var count = 0

for num in nums {

if count == 0 {

candidate = num

count = 1

} else if candidate == num {

count += 1

} else {

count -= 1

}

}

return candidate!

}

}

// Пример использования:

let solution = Solution()

let nums = [3, 3, 4, 2, 4, 4, 2, 4, 4]

print(solution.majorityElement(nums)) // Output: 4

Это решение имеет временную сложность O(n) и использует константное количество дополнительной памяти, что соответствует требованию O(1).

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Задача возвращает максимальную прибыль, которую вы можете получить от этой сделки . Например: дан массив цены [7, 1, 5, 3, 6, 4]**

**Покупайте в день 2 (цена = 1) и продавайте в день 5 (цена = 6), прибыль = 6-1 = 5.**

**Обратите внимание, что покупка во второй день и продажа в первый день не разрешены, поскольку вы должны купить, прежде чем продавать. Если мы не можем получить никакой прибыли, возвращаем 0.**

class Solution {

func maxProfit(\_ prices: [Int]) -> Int {

if prices.isEmpty {

return 0 // Возвращаем 0, если массив пустой

}

var maxProfit = 0

var minPrice = prices[0] // Изначально устанавливаем минимальную цену как первый элемент массива

for price in prices {

minPrice = min(minPrice, price) // Обновляем минимальную цену, если текущая цена меньше

maxProfit = max(maxProfit, price - minPrice) // Обновляем максимальную прибыль, если текущая прибыль больше

}

return maxProfit

}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Алгоритм бинарного поиска:**

// Note: If you aren’t familiar with Comparable, please check out “Generics” chapter in Swift book

func binarySearch <T: Comparable>(\_ list: [T], item: T) -> Int? {

// low and high keep track of which part of the list you'll search in.

var low = 0

var high = list.count - 1

// While you haven't narrowed it down to one element ...

while low <= high {

//... check the middle element

let mid = (low + high) / 2

let guess = list[mid]

// Found the item.

if guess == item {

return mid

}

// The guess was too high.

if guess > item {

high = mid - 1

} else {

low = mid + 1

}

}

return nil

}

let myList = [1, 3, 5, 7, 9]

print(binarySearch(myList, item: 3) ?? "Not Found") // => 1

print(binarySearch(myList, item: -1) ?? "Not Found") // => Not Found

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Функция для поиска наименьшего элемента массива:**

// Finds the smallest value in an array

func findSmallestIndex <T: Comparable> (\_ arr: [T]) -> Int {

// Stores the smallest value

var smallest = arr[0]

// Stores the index of the smallest value

var smallestIndex = 0

for i in 1..<arr.count {

if arr[i] < smallest {

smallest = arr[i]

smallestIndex = i

}

}

return smallestIndex

}

**На основе это функции можно написать функцию сортировки выбором:**

// Sort array

func selectionSort <T: Comparable> (arr: [T]) -> [T] {

// We don't need any calculation if the array length is 1

guard arr.count > 1 else { return arr }

var newArr: [T] = []

// We have to make mutableArray reference copy of original array, because Swift 3 doesn't allow to get var parameter

var mutableArr = arr

for \_ in 0..<mutableArr.count {

//Finds the smallest element in the array and adds it to the new array

let smallestIndex = findSmallestIndex(mutableArr)

newArr.append(mutableArr.remove(at: smallestIndex))

}

return newArr

}

print(selectionSort(arr: [5, 3, 6, 2, 10])) // => [2, 3, 5, 6, 10]